Designing in UML With the Team Software Process

David R. Webb, Ilya Lipkin, and Evgeniy Samurin-Shraer
309 Software Maintenance Group

The Team Software Process℠ (TSP℠) is a good project management tool that enforces a disciplined approach to software engineering, drastically improving cost and schedule performance and the production of quality products. One of the ways TSP improves product quality is through emphasis on design. A heavy design emphasis is also the hallmark of the newer programming environments. This article examines how modern design techniques can be used on a TSP project.

In late 2004, the 309 Software Maintenance Group (309 SMXG) at Hill Air Force Base took on a series of new projects. One of these projects focused on updating software in an embedded weapon system, a task that 309 SMXG was proficient in performing; however, this new system had been developed using a modern Unified Modeling Language (UML) auto code generation toolset, with which the group had no prior experience. Assessed as a Capability Maturity Model® (CMM®) Level 5 organization in 1998, and focusing on a CMM Integration℠ (CMMI®) assessment in 2006, the 309 SMXG was confident in its ability to deliver software on time and within budget for traditional projects, but determined that this newer system required a different approach.

With this in mind, 309 SMXG brought in an entirely new team of UML developers to work on the maintenance of the new weapon system. The obvious drawback to this approach was that this new team had little or no experience using the disciplined software engineering techniques required by the CMM and CMMI. Internal group policies, which the team was required to follow, demanded the project to tailor processes from the organizational standard and to follow the CMMI's specific practices for everything from project planning to quantitative project management. However, due to the inexperience of the team members, the project was struggling to come up to speed on these practices on a very short schedule.

Having had success in the past using the Software Engineering Institute's (SEI℠'s) Team Software Process℠ (TSP℠) [1] to bolster the group's own internal processes for newer teams, the 309 SMXG decided to use this approach on the new project. The resulting marriage between UML and TSP created a flexible and mobile design tool with a rigid and disciplined process.

TSP and CMMI

Simply put, TSP is CMM/CMMI Level 5 at a project level. It is supported by team members who perform Level 5 practices at a personal level using SEI's Personal Software Process℠ (PSP℠). A recent report by the SEI indicates that adopting the TSP will satisfy most of the specific practices of the CMMI process areas [2]. To quickly bring the team up to speed on the CMMI, the 309 SMXG put the entire software team through PSP training, which required about six weeks. They also trained the team in using a PSP tracking tool called the Process Dashboard that automates many of the personal planning and tracking activities required by the PSP.

> *"The resulting marriage between UML and TSP created a flexible and mobile design tool with a rigid and disciplined process."*

At the conclusion of this training, a certified SEI TSP launch coach took the team through a one-week TSP launch session. These sessions are used to determine stakeholder goals, establish team roles and processes, and produce detailed earned value, quality, and risk management plans. Since these are all key elements of the CMMI, the launch sessions are vital to the disciplined software engineering practices required by the model.

It was during this launch that the team encountered its first issues with the project's UML environment. To create the detailed earned value plans required by the TSP, each of the team's major tasks required some type of size criteria of task development for estimating purposes. In other words, the team needed a way of determining which tasks were larger than others and calculating how much effort those tasks would require. In a traditional software environment, the team would have estimated using source lines of code (SLOC) and converted from SLOC to effort using a team productivity rate. There were two problems with this traditional approach: (1) since the team was new, there was a lack of historical data upon which to base productivity estimates, and (2) the auto-generated code features of the UML environment made traditional size estimation very problematic. Since each of the team members had measured their effort and lines of code in the PSP class, the issue of productivity could have been addressed by using classroom averages of SLOC/hour; however, the problem of size estimation proved much more difficult.

UML Design With TSP

UML is a *language* developed by Grady Booch, James Rumbaugh, and Ivar Jacobson that uses object-oriented concepts and methodologies to model software systems [3]. Simply stated, UML consists of a set of diagrams that allow designers to examine a software program from several different points of

Table 1: *PSP Design Template Structure (SEI)*

Object Specification	Internal	External
Static	Logical Specification Template	Functional Specification Template
Dynamic	State Specification Template	Operational Scenario Template

® Capability Maturity Model, CMM, and CMMI are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
℠ CMM Integration and SEI are service marks of Carnegie Mellon University.

view prior to creating the code [4]. The standard set of UML diagrams is the following:

- Use Cases.
- Class Diagrams.
- Object Diagrams.
- Sequence Diagrams.
- Collaboration Diagrams.
- State-Chart Diagrams.
- Activity Diagrams.
- Component Diagrams.
- Deployment Diagrams.

Combined views of these diagrams create a complete description of the software design.

As it turns out, the UML view of design does not differ significantly from the basic design techniques taught during the PSP course. In fact, the PSP design templates and design scripts provide a clear and concise description of steps needed to produce an effective design in UML. The PSP design-first technique uses four *orthogonal* views of any software design: internal static (module or part construction such as the logical layout of a module), internal dynamic (characteristic based upon changing values within the module), external static (the relationship of a module or part to other parts in the product), and external dynamic (the interactions this part or module has with other parts in the product). Each design view has a template to capture the information (see Table 1). The following is an example of how to use them [5].

Let us assume we want to develop software for a standard traffic light. The traffic light has three possible conditions: red (stop), yellow (caution), and green (go). The Operational Scenario Template (external/dynamic view) is used to capture the fact that the light consistently changes from green to yellow to red based solely upon a timed sequence. The State Specification Template (internal/dynamic view) captures the fact that there is a defined set of states through which the traffic light moves: green, to yellow, to red, and back to green again. It is not possible to go from green to red or yellow to green (see Table 2).

Now that requirements for the traffic light have been presented, it is time to capture them in the design. To perform this task, it is best to use a set of templates on the requirements; several templates will be used.

1. Operational Scenario Template (dynamic/external) is the system requirements, which are treated as use cases for the traffic light.

State #, Name	Description	Attributes
Initial	System not running	Timer event set to 30 seconds
Green	On timer time out event go to(goYellow) Yellow	Timer event set to 45 seconds
Yellow	On timer time out event go to(goRed) Red	Timer event set to 10 seconds
Red	On timer time out event go to(goGreen) Green	Timer event set to 30 seconds

Table 2: *Example of the State Specification Template (SEI)*

Object Specification	Internal	External
Static	Class Diagrams	Component Diagrams Deployment Diagrams Object Diagrams
Dynamic	Activity Diagrams State-Chart Diagrams	Use Cases Sequence Diagrams Collaboration Diagrams

Table 3: *PSP Design Structure for UML (SEI)*

2. Functional Specification Template (static/external) is used to describe the traffic light timer functionality and how it is used.

"... the PSP design templates and design scripts provide a clear and concise description of steps needed to produce an effective design in UML."

3. State Specification Template (dynamic/internal) is used to capture the flow of events between states that are now colors of the traffic light (see Table 2).
4. Logic Specification Template (static/internal) can then be used to capture steps in pseudo code for the user-entered, action-code portion of UML. This template makes even the traditional process of coding almost trivial. For this example, there is no pseudo code needed as it is done entirely in UML events.

When all of these templates are completed, it becomes obvious how the logic must be constructed for the software to run the traffic light system design. It is now a simple task to draw the design in UML. The UML design then can auto-generate code at this stage (Figure 1).

Final UML designs produced, as shown in Table 3, fit into the same four quadrants as the PSP design templates. In Figure 1, the design description of the traffic light is captured using UML techniques. The PSP template used to implement this design description is the State Specification template, which is equivalent to the UML State-Chart Diagram.

The only other diagram created and required to complete this design is from

Figure 1: *Example of the Working UML Design Solution for Traffic Light State-Chart Diagram*

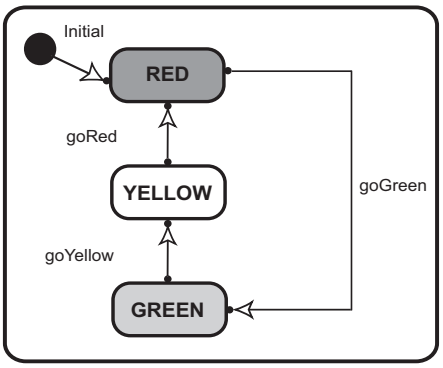
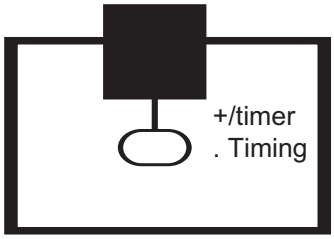


Figure 2: *Example of the Working UML Design Solution for Traffic Light Collaboration (Structure) Diagram*



the Functional Specification Template. This template describes timer functionality to the traffic light system, which translates into a collaboration diagram (Figure 2).

Using UML does not contradict the TSP philosophy, but in fact works quite nicely with this highly disciplined process. In addition, some UML tools (such as the IBM Rational Rose Real Time suite, used by the 309 SMXG project) offer the unique ability to utilize these diagrams to create auto-generated code. The chief reason that size estimation was so difficult for this new 309 SMXG project was due to auto-generation capability of UML. This seemed to be the only real issue in dealing with a modern UML design and development environment on a TSP project.

A New Size Metric

The problem with using a typical LOC counter to determine size was the auto-generated code. When drawing a single new line in a diagram (what you might think of as a single SLOC change), could (and often did) generate dozens of new and changed SLOC. This was due to the fact that the tool would *rethink* the entire software module, rather than just the single function being addressed. The result was that there existed little or no correlation between SLOC generated and effort required to produce the change. A traditional SLOC estimate would mean little or nothing to this team; something else was needed for size and effort estimation.

The size metric required for this project had to meet two criteria to be usable for estimating and tracking: (1) it had to correlate to the work performed, and (2) it had to be automatically measurable – counting any measurement by hand was deemed to be too slow and inaccurate. After a great deal of research, the 309 SMXG team decided to try out a rough approximation of engineer-generated and auto-generated SLOC. Using what limited data they had, the team's design manager determined that in an average build, approximately one-third of the code was *user developed* (i.e., directly correlated to the work performed). The remaining two-thirds was auto-generated with considerably less effort on the engineer's part. With this in mind, the team examined modules created by the original development team prior to the maintenance phase and applied these findings to determine *adjusted SLOC*. Modules were then estimated using Adjusted SLOC and the estimates converted to effort using PSP classroom productivity rates [6].

Once these metrics were determined, sizes were estimated and effort computed; this led to the production of the detailed earned value plan that is the hallmark of TSP projects.

Running the TSP Project

Since the metrics used to estimate and track the project size and effort were new, there was obviously some concern about the accuracy of the estimates. The TSP practice of tracking progress at the personal level and then *rolling* that data up to the team level allowed this project to find and correct potential estimate issues before the project missed any schedule deadlines. Effectively, the TSP and the use of the Process Dashboard kept members of the team on task and on schedule [7].

The process steps needed to complete the design were captured as earned value tasks on each individual software engineer's dashboard. The tasks were then broken down further into something that could be accomplished on a weekly basis. During product execution,

Table 4: *High Level Design Process Tasks*

Entry	Task	Exit

• Need to create or modify system design
• Project plans approved
• Resources identified and available
• Customer requirements configured
	• Research
	  • Gather data/documentation
	  • Identify assumptions
	• Analyze
	  • Analyze/functionally decompose requirements
	  • Perform risk analysis
	  • Identify required system resources (throughput, speed, memory, etc.)
	• Consider alternate solutions
	  • List the make/buy/reuse alternatives
	  • List other alternatives
	  • Utilize decision analysis and resolution as required
	  • Select solution(s)
	• Determine/design interfaces
	• Create design document(s)
	• Determine and document test requirements
	• Perform personal review
	• Hold peer reviews/perform corrective actions
	• Configure outputs in accordance with the project's configuration management plan
	• Conduct preliminary design review (PDR)
	  • Coordinate with relevant stakeholder
	Verification and Validation
	• Perform personal review
	• Hold peer reviews/perform corrective actions
	• Acceptance of preliminary design by the interdisciplinary team and the customer
	• Relevant stakeholders participate in creation of outputs and accept the preliminary design
	• Design simulation
	• PDR
		• Preliminary design completed
		• PDR completed
		• Record project data (effort, schedule, risk, defect data, minutes, lessons learned, etc.)

the team met weekly to ascertain both team and individual earned value. If a certain task was taking longer than expected, it became highly visible to the team members and the project manager during the weekly meeting. Any problem tasks were discussed during weekly team meetings and addressed by either reassigning it to another member or by addressing the scope with the customer early enough so as to not cause a schedule slip. In addition, these tasks were used to determine if some aspects of the design had been underestimated or overestimated for the development of the software. TSP allowed the team to catch problems at the first sign of an anomaly and to address it quickly, without significant cost to the customer or the project [7].

Furthermore, these tasks were based simply on the steps defined in the project that were tailored from the organizational process required by CMMI (see Table 4). The process descriptions (called *scripts* in the TSP) served as handy references for the team members, detailing what needed to be followed and completed on the dashboard tracking tool. The breakdown of the processes through the dashboard allowed for the design steps to be tracked individually.

One issue the team did uncover was a lack of dependency tracking. For example, if Team Member A needed Task 1 to be completed by Team Member B prior to working his/her task, then Team Member A's schedule is dependent upon Team Member B; however, this dependency is not reflected in the earned value plan created by the team during the initial launch.

This is due to the fact that the TSP earned value plan does not identify these dependencies. The team found that this is both a weakness and strength of the TSP. While the lack of proper dependency tracking often causes confusion and could result in inaccurate project status, TSP earned value tracking allows tasks to be worked in any order. As a result, team members are free to work on other issues (clearly identified in their plan) while they are awaiting the completion of a dependent task. The danger is that team members can also choose to complete all other tasks assigned to them first and leave the dependency task for last. This, in effect, creates dead time for other team members who depend on the incomplete task. The team solved this problem by closely coordinating with each other during the weekly team meetings [8].

The 309 SMXG project found that it was possible to run a TSP project within a UML environment.

Lessons Learned

After completion of the design, the lessons learned from implementation of the TSP include the following:

- Although the adjusted SLOC estimation worked for first pass through the project, it has since been found that it does not always correlate well to effort. The reason for this is that the UML tool does not consistently convert code in the *one-third user code* ratio presented above. The UML tool is highly dependent on the *whims* of the auto-generator when converting UML to SLOC. In future TSP iterations, the team has determined to find a new method of code counting and estimating that more accurately reflects the effort and time spent on UML. To find this new method, detailed statistical data is being gathered that reflects UML design objects and effort taken to produce them. In the meantime, various size metrics are being determined and gathered for code size proxy [8].
- The team determined that TSP does work in a UML auto-generated code environment, as long as size estimation issues are properly dealt with and the percent of time spent in each phase (design, code, test, etc.) is adjusted to increase the amount of time needed to design.
- When designing using UML and auto-generated code, part of the design may also be considered *implementation* or *coding*. As a result, the typical PSP phases must be modified to reflect that the design phase will now share tasking from the coding phase. On our project, this resulted in a separation of the code phase into two parts: design/code and code. The design/code reflected the auto-generated part of the UML, and the code reflected the user-entered portion of the UML implementation.

"The TSP practice of tracking progress at the personal level and then rolling that data up to the team level allowed this project to find and correct potential estimate issues before the project missed any schedule deadlines."

Conclusion

The TSP processes were very effective for this team. Not only did the introduction of the TSP bring the team's CMMI compliance quickly to Level 3 and beyond, but the structure and format of the processes allowed for better understanding of each team members' responsibilities and tasks involved in completion of the design project.◆

References

1. Webb, David R., and Watts Humphrey. "Using the TSP on the TaskView Project." CROSSTALK Feb. 1999 <www.stsc.hill.af.mil/crosstalk/1999/02/index.html>.
2. McHale, James, and Daniel S. Wall. "Mapping TSP to CMMI." Pittsburgh, PA: Software Engineering Institute, 22 June 2005.
3. Booch, Grady, James Rumbaugh, and Ivar Jacobson. The Unified Modeling Language User Guide. Addison-Wesley, 2001.
4. Sanderfer, Lynn. "How and Why to Use the Unified Modeling Language." CROSSTALK June 2005 <www.stsc.hill.af.mil/crosstalk/2005/06/0506sanderfer.html>.
5. Humphrey, Watts S. A Discipline for Software Engineering. Addison-Wesley, 1995.
6. Tuma, David, and David R. Webb. "Personal Earned Value: Why Projects Using the Team Software Process Consistently Meet Schedule Commitments." CROSSTALK Mar. 2005 <www.stsc.hill.af.mil/crosstalk/2005/03/0503tuma.html>.
7. Webb, David R. "All the Right Behavior." CROSSTALK Sept. 2002 <www.stsc.hill.af.mil/crosstalk/2002/09/webb.html>.
8. Webb, David R. "Managing Risk With TSP." CROSSTALK June 2000 <www.stsc.hill.af.mil/crosstalk/2000/06/webb.html>.

About the Authors

David R. Webb is a senior technical program manager for the Software Division of Hill Air Force Base in Utah, a Capability Maturity Model® Level 5 organization. He is a project management and process improvement specialist with more than 18 years of technical, program management, and process improvement experience with Air Force software. Webb is a Software Engineering Institute-authorized instructor of the Personal Software Process℠, a Team Software Process℠ launch coach, and has worked as an Air Force section chief, systems software engineer, and test engineer. Webb has a bachelor's degree in electrical and computer engineering from Brigham Young University.

7278 4th ST
BLDG 100 RM 109
Hill AFB, UT 84056
Phone: (801) 940-7005
DSN: 940-7005
E-mail: david.webb@hill.af.mil

Ilya Lipkin is an electronics engineer at the 309th Software Maintenance Group at the Ogden Air Logistics Center, Hill Air Force Base, Utah. His current research interests include artificial intelligence, human knowledge capture and analysis, neural networks, fuzzy logic, user interface design, software engineering, and customer relations management. Lipkin has a Bachelor of Science in computer engineering from the University of Toledo, a Master of Science in computer engineering from the University of Michigan, and is a doctoral candidate at the University of Toledo Business School.

7278 4th ST
BLDG 100 RM 109
Hill AFB, UT 84056
Phone: (801) 586-4477
Fax: (801) 586-2042
E-mail: ilya.lipkin@hill.af.mil

Evgeniy Samurin-Shraer is an electrical engineer at the 309th Software Maintenance Group at the Ogden Air Logistic Center, Hill Air Force Base, Utah. His current research interests include antenna design, resonance frequency circuits, and application of the microwave theory to the biomedical problems. Samurin-Shraer has a Bachelor of Science in electrical engineering and a Master of Science in electrical engineering from the University of Toledo.

7278 4th ST
BLDG 100 RM 109
Hill AFB, UT 84056
Phone: (801) 586-2048
Fax: (801) 586-2042
E-mail: evgeniy.samurin-shraer@hill.af.mil

Letter to the Editor

Dear CrossTalk Editor,
Kevin Stamey opened the November 2005 issue with these remarks in his "From the Sponsor" column:

> ... other engineering design disciplines have been in place for centuries; however, software engineering is still relatively new. The discipline of software design has only been matured for a few decades. It wasn't until the 1960s that the first software products hit the marketplace ... Our dominant programming language, C++, didn't emerge until the 1980s ...

The relative newness of software engineering is often cited when explaining the frustrations of the ongoing software crisis. However, the fact that current practices have only been around for a few decades, is that really extraordinary? Is our phenomenal growth all that unique? And have all the other engineering disciplines really been around for centuries?

Aeronautical and aerospace engineering may not be as new as software engineering, but there are certainly not centuries of experience in those fields. Not long ago, most aircraft were propeller-driven, and we referred to the sound barrier. Electrical engineering cannot be considered a centuries-old discipline unless you start with Ben Franklin's kite-and-key experiments.

Many of software engineering's principal tools have indeed been in place for a relatively short time, but isn't that true of most engineering disciplines? Niels Bohr's simplistic model of the atom is less than 100 years old. Physicists are continually discovering new particles; researchers are only beginning to explore the possibilities of quantum computing. Huge advances have been made by materials scientists, meaning circuitry and silicon technologies have undergone several significant advances in a relatively short time.

Indeed, our newness presents some formidable challenges, and provides fodder for intense debate. But we ought to avoid emphasizing that this newness makes us unique, or that our needing to adapt to rapidly evolving technologies and standards is somehow exclusive. Such naiveté presents us as making excuses for our shortcomings rather than boldly confronting challenges.

The first transistor was fabricated in the 1940s, and the first rudimentary integrated circuits were fabricated in the 1950s, about the same time that early compilers came into being. Software engineers don't need more time for their field to mature; like others in technological and engineering fields, we are challenged to advance and progress in a disciplined yet rapid fashion to keep up with the monumental advances occurring in the world around us.

There are several aspects of software engineering that set us apart from other engineering disciplines. Most notably, our end product is tied to the virtual world, not the physical world. As such, our discipline is governed less by the laws of physics, and we don't rely on equations as fundamental, foundational truths. This makes it harder to build upon the previous work of theoreticians in a predictable way – something that I think better explains our slow maturation than our relative newness.

John Reisner
Air Force Institute of Technology
<jreisner@gimail.af.mil>